

Assistant Commissioner for Patents
Washington, D.C. 20231
Sir:

1-28-00
ATTORNEY DOCKET: Y0999-357 (8728-320)
Date: January 27, 2000
Express Mail Label: EL433927938US
Date of Deposit: January 27, 2000

A

Transmitted herewith for filing is the Patent Application of:

Inventors: Michael K. Gschwind

For: METHODS FOR RENAMING STACK REFERENCES IN A COMPUTER PROCESSING SYSTEM

Enclosed are: [X] 40 sheets of specification; [X] 1 sheet(s) of Abstract; [X] 12 sheet(s) of claims; [X] 6 sheet(s) of drawing(s);

- [X] An assignment of the invention to International Business Machines Corporation with Recordation Form.
- [X] Declaration and Power of Attorney.
- [] A certified copy of a _____ application, from which priority under Title 35 USC §119 is claimed.
- [X] Associate Power of Attorney.

The filing fee has been calculated as shown below:

| (Col. 1) | | (Col. 2) | OTHER THAN A SMALL ENTITY | |
|---|-----------|-----------|------------------------------|------------|
| FOR: | NO. FILED | NO. EXTRA | RATE | FEE |
| BASIC FEE | | | | \$690.00 |
| TOTAL CLAIMS | 39 - 20 = | 19 | X \$18 = | 342.00 |
| INDEP CLAIMS | 4 - 3 = | 1 | X \$78 = | 78.00 |
| MULTIPLE DEPENDENT CLAIMS PRESENTED | | | + 260 = | |
| If the difference in Col. 1 is less than zero, enter "0" in Col. 2. | | | TOTAL | \$1,110.00 |

JC584 U.S. PTO
09/492544
01/27/00

[] A check in the amount of \$_____ to cover the [] filing fee(s), [] recording fee is enclosed.

[X] Please charge my Deposit Account No. 50-0510/IBM (Yorktown Heights) in the amount of \$1,110.00.

[X] The Commissioner is hereby authorized to charge payment of the following fees associated with this communication or credit any overpayment to Deposit Account No. 50-0510/IBM (Yorktown Heights). A duplicate copy of this sheet is enclosed.

[X] Any additional filing fees required under 37 CFR 1.16.

[X] Any patent application processing fees under 35 CFR 1.17.

Respectfully submitted,

By:

Gaspere J. Randazzo
Gaspere J. Randazzo
Registration No. 41,528

Please address all
correspondence to:

F. CHAU & ASSOCIATES, LLP
1900 Hempstead Tpke., Suite 501
East Meadow, NY 11554
Tel: (516) 357-0091
Fax: (516) 357-0092

Attorney for:

IBM Corporation
Intellectual Property Law Dept.
P.O. Box 218
Yorktown Heights, NY 10598

CERTIFICATION UNDER 37 C.F.R. §1.10

I hereby certify that this Application transmittal and documents referred to as enclosed are being deposited with the United States Postal Service on this date January 27, 2000 in an envelope as "Express Mail Post Office to Addressee" Mailing Label Number EL433927938US addressed to: Assistant Commissioner for Patents, Box Patent Application, Washington, D.C. 20231.

Gaspere J. Randazzo
Gaspere J. Randazzo

JC584 U.S. PTO
01/27/00

Please type a plus sign (+) inside this box → ☐

PTO/SB/05 (4/98)
Approved for use through 09/30/2000. OMB 0651-0032
Patent and Trademark Office: U.S. DEPARTMENT OF COMMERCE
Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

UTILITY PATENT APPLICATION TRANSMITTAL

(Only for new nonprovisional applications under 37 C.F.R. § 1.53(b))

Attorney Docket No. Y0999-357 (8728-320)
First Inventor or Application Identifier GSCHWIND
Title METHODS FOR RENAMING STACK REFERENCES IN..
Express Mail Label No. EL433927938US

APPLICATION ELEMENTS

See MPEP chapter 600 concerning utility patent application contents.

1. ☒ * Fee Transmittal Form (e.g., PTO/SB/17)
(Submit an original and a duplicate for fee processing)
2. ☒ Specification [Total Pages 53]
(preferred arrangement set forth below)
 - Descriptive title of the invention
 - Cross References to Related Applications
 - Statement Regarding Fed sponsored R & D
 - Reference to Microfiche Appendix
 - Background of the invention
 - Brief Summary of the invention
 - Brief Description of the Drawings (if filed)
 - Detailed Description
 - Claim(s)
 - Abstract of the Disclosure
3. ☒ Drawing(s) (35 U.S.C. 113) [Total Sheets 6]
4. Oath or Declaration [Total Pages 2]
 - a. ☒ Newly executed (original or copy)
 - b. ☐ Copy from a prior application (37 C.F.R. § 1.63(d))
(for continuation/divisional with Box 16 completed)
 1. ☐ DELETION OF INVENTOR(S)
Signed statement attached deleting
inventor(s) named in the prior application,
see 37 C.F.R. §§ 1.63(d)(2) and 1.33(b).

* NOTE FOR ITEMS 1 & 13: IN ORDER TO BE ENTITLED TO PAY SMALL ENTITY
FEES, A SMALL ENTITY STATEMENT IS REQUIRED (37 C.F.R. § 1.27), EXCEPT
IF ONE FILED IN A PRIOR APPLICATION IS RELIED UPON (37 C.F.R. § 1.28).

ADDRESS TO: Assistant Commissioner for Patents
Box Patent Application
Washington, DC 20231

5. ☐ Microfiche Computer Program (Appendix)
6. Nucleotide and/or Amino Acid Sequence Submission
(if applicable, all necessary)
 - a. ☐ Computer Readable Copy
 - b. ☐ Paper Copy (identical to computer copy)
 - c. ☐ Statement verifying identity of above copies

ACCOMPANYING APPLICATION PARTS

7. ☒ Assignment Papers (cover sheet & document(s))
8. ☐ 37 C.F.R. § 3.73(b) Statement (when there is an assignee) ☒ Power of Attorney
9. ☐ English Translation Document (if applicable)
10. ☒ Information Disclosure Statement (IDS)/PTO-1449 ☒ Copies of IDS Citations
11. ☐ Preliminary Amendment
12. ☒ Return Receipt Postcard (MPEP 503)
(Should be specifically itemized)
13. ☐ * Small Entity Statement(s) ☐ Statement filed in prior application.
(PTO/SB/09-12) Status still proper and desired
14. ☐ Certified Copy of Priority Document(s)
(if foreign priority is claimed)
15. ☒ Other: ASSOCIATE POWER OF ATTORNEY

16. If a CONTINUING APPLICATION, check appropriate box, and supply the requisite information below and in a preliminary amendment:
☐ Continuation ☐ Divisional ☐ Continuation-in-part (CIP) of prior application No: _____
Prior application information: Examiner _____ Group / Art Unit: _____

For CONTINUATION or DIVISIONAL APPS only: The entire disclosure of the prior application, from which an oath or declaration is supplied under Box 4b, is considered a part of the disclosure of the accompanying continuation or divisional application and is hereby incorporated by reference. The incorporation can only be relied upon when a portion has been inadvertently omitted from the submitted application parts.

17. CORRESPONDENCE ADDRESS

☐ Customer Number or Bar Code Label

(Insert Customer No. or Attach bar code label here)

or ☐ Correspondence address below

Name Gaspare J. Randazzo
Address F. Chau & Associates, LLP
1900 Hempstead Turnpike, Suite 501
City East Meadow State New York Zip Code 11554
Country USA Telephone 516-357-0091 Fax 516-357-0092

Name (Print/Type) Gaspare J. Randazzo Registration No. (Attorney/Agent) 41,528
Signature *Gaspare J. Randazzo* Date 1/27/2000

Burden Hour Statement: This form is estimated to take 0.2 hours to complete. Time will vary depending upon the needs of the individual case. Any comments on the amount of time you are required to complete this form should be sent to the Chief Information Officer, Patent and Trademark Office, Washington, DC 20231. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. SEND TO: Assistant Commissioner for Patents, Box Patent Application, Washington, DC 20231.

METHODS FOR RENAMING STACK REFERENCES IN A COMPUTER PROCESSING SYSTEM

BACKGROUND

5 1. Technical Field

The present invention generally relates to computer processing systems and, in particular, to methods for renaming stack references in a computer processing system.

10 2. Background Description

15 A memory serves as a repository of information in a computer processing system. FIG. 1 is a block diagram illustrating a typical layout of a memory 100 of a computer program according to the prior art. The layout consists of distinct memory areas, including a program text area 104, a program data area 106, a heap 108, and a program stack 110. Program text area 104 is used to store program text (i.e., computer instructions). Program data area 106 is used to store program data (for static data references). Heap 108 is used for dynamically allocated objects and program stack 110 is used for function-local variables.

As shown, memory 100 stores different types of data in distinct memory areas. The following different mechanisms are used to access these memories:

1. Program text area 104 stores the computer instructions describing the actions of a program, and possibly program constants. Program text area 104 is usually read-only and accessed using the program counter.
2. Program data area 106 holds static data references, e.g., global program variables. Program data area 106 is accessed using either a global data pointer or a table of contents data structure.
3. Heap 108 holds dynamically allocated objects and is accessed using pointers held in any of the processor registers.
4. Program stack 110 usually holds function-local variables and is accessed using special-purpose registers, such as the stack pointer (SP), frame pointer (FP), or argument pointer (AP).

Usually, all program memory can be accessed through the use of pointers which are stored in a register.

However, the access mechanisms described above are generally used for each area in typical programs.

In general, a processor accesses information from the memory, performs computations thereon, and stores the results back to memory. Unfortunately, memory access incurs a number of costs. A description of some of these costs will now be given.

When a memory access operation is first detected, the address to be accessed must be resolved. Moreover, the registers employed for the address computation must be available.

If the processor wants to reorder memory read operations with respect to other memory operations, and it cannot be determined that the read addresses are different at the time when they are to be reordered, then checks for memory address ambiguities need to be performed.

In addition, since store operations modify the processor state, they typically have to be performed in-order. This causes further slowdowns in achievable processor performance by serializing operations when multiple live ranges are assigned to the same memory location. Thus, limitations are typically imposed on the degree of reordering that can be performed in a superscalar

processor, when multiple independent values are assigned to the same memory address.

Moreover, load and store operations typically require access to a cache(s). However, accessing a cache is slower in comparison to accessing processor registers, which represent a higher level in the memory hierarchy of a computer processing system.

Many of the serializing effects of memory references result from the way in which programs are written by programmers. However, serializing effects of memory references may also result from the way programs are translated from their source level representation to the actual machine. In such a case, references are made to the program stack.

The program stack stores stack frames, that is, records containing the values for local variables of functions, as well as parameters passed between functions. Stack locations are reused frequently, with different functions using memory locations with the same address to store unrelated objects.

Consider the following example code written in the C programming language:

```

int mult3 (int a)
{
    return a * 3;
}

5
int inc (int b)
{
    return b+1;
}

10
int compute(int a, int b)
{
    int tmp1, tmp2;

    tmp1 = mult3(a);
    tmp2 = inc(b);

    return tmp1+tmp2;
}

20

```

When this code is translated to Intel x86 machine code, the following instructions will be generated:

```

25 1      mult3:
2      imull $3,4(%esp),%eax
3      ret
4
5      inc:
6      movl 4(%esp),%eax
30 7      incl %eax
8      ret
9
10     compute:
11     pushl %esi
12     pushl %ebx
35 13     movl 12(%esp),%eax
14     movl 16(%esp),%ebx
15     pushl %eax
16     call mult3
40 17     addl $4,%esp

```

```

18      movl %eax,%esi
19      pushl %ebx
20      call inc
21      addl $4,%esp
5  22      addl %esi,%eax
23      popl %ebx
24      popl %esi
25      ret

```

10 The immediately preceding code illustrates several examples of the inefficiencies of holding the processor stack in memory:

- 15 1. The values of registers ESI and EBX are stored on the stack at instructions 11 and 12, and restored at instructions 23 and 24. These values could have been held in processor-internal registers.
- 20 2. The parameters a and b which were pushed onto the stack by the calling function must be read from the stack into a processor register, and then stored on the stack for functions mult3 and inc, respectively.
- 25 3. The parameters a and b for functions mult3 and inc, respectively, are stored at the same stack location, so operations from function inc cannot be scheduled at the same time as the instructions for function mult3. This serialization is not necessary.

The serializing effects of stack references due to the reuse of memory locations and the manipulation of the stack pointer is described by Postiff et al., in "The Limits of Instruction Level Parallelism in SPEC95 Applications", International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII) Workshop on Interaction between Compilers and Computer Architectures (INTERACT-3), October 1998. Postiff et al. further describe the performance improvements which can be achieved by resolving these serializing effects.

3. Problems with the State of the Art

It is to be appreciated that previous memory renaming has been based on renaming of general memory references, and has tended to ignore multiprocessor effects. Some of these prior art approaches will now be described.

It is to be appreciated that memory renaming typically includes the prediction of data dependencies. A mechanism to predict data dependencies dynamically without computation of the address is described by A. Moshovos and G. Sohi, in "Streamlining Inter-operation Memory Communication via Data Dependence Prediction", Proceedings of 30th Annual International Symposium on Microarchitecture Research

Triangle Park, NC, December 1997. Predicting dependencies is necessary because the addresses of load and store operations may be unresolved. To ensure correctness of the predictions, these memory operations need to be eventually executed. A similar approach for predicting dependencies is described by G. Tyson and T. Austin, in "Improving the Accuracy and Performance of Memory Communication Through Renaming", Proceedings of 30th Annual International Symposium on Microarchitecture Research, Triangle Park, NC, December 1997. Moshovos & Sohi and Tyson & Austin provide a general technique for promoting accesses to memory into processor-internal registers. This requires hardware of significant complexity. Moreover, prediction is used, which is not as accurate as actual decoding of the instruction, and may require expensive repair actions. An address resolution buffer which supports out-of-order execution of memory operations and memory renaming is described by M. Franklin and G. Sohi, in "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References", IEEE Transactions on Computers, Vol. 45, No. 5, May 1996. Disadvantageously, this buffer is expensive, the required hardware is complex, and the buffer does not consider multiprocessor systems and their consistency requirements.

U.S. Patent No. 5,911,057, entitled "Superscalar
Microprocessor Having Combined Register and Memory Renaming
Circuits, Systems, and Methods", issued on June 8, 1999, the
disclosure of which is incorporated herein by reference,
5 describes an architecture for renaming memory and register
operands in uniform fashion. Memory coherence is based on
"snooping" memory requests. While this approach is
sufficient for the in-order execution of memory operations
in a multiprocessor computing system, out-of-order operation
10 in a multiprocessor system may generate incorrect results.
U.S. Patent No. 5,838,941, entitled "Out-of-order
Superscalar Microprocessor with a Renaming Device that Maps
Instructions from Memory to Registers", issued on November
17, 1998, the disclosure of which is incorporated herein by
15 reference, describes symbolic renaming of memory references.
This invention deals with equivalence of all types, and
requires lockup of an associative array to establish
equivalence between expression and names. This results in a
complex architecture with potentially severe cycle time
20 impact.

Thus, it would be desirable and highly
advantageous to have a method for eliminating serializing
effects resulting from stack references. It would be

further desirable and advantageous if such method was applicable in a multiprocessor system.

SUMMARY OF THE INVENTION

5 The problems stated above, as well as other related problems of the prior art, are solved by the present invention, methods for renaming stack references in a computer processing system.

10 According to a first aspect of the invention, there is provided a method for renaming memory references to stack locations in a computer processing system. The method includes the steps of detecting stack references that use architecturally defined stack access methods, and replacing the stack references with references to processor-internal registers.

15 According to a second aspect of the invention, the method further includes the step of synchronizing an architected state between the processor-internal registers and a main memory of the computer processing system.

20 According to a third aspect of the invention, the method further includes the step of inserting in-order write operations for all of the stack references that are write stack references.

According to a fourth aspect of the invention, the method further includes the step of performing a consistency-preserving operation for a stack reference that does not use the architecturally defined stack access methods.

According to a fifth aspect of the invention, the step of performing a consistency-preserving operation includes the step of bypassing a value from a given processor-internal register to a load operation that references a stack area and that does not use the architecturally defined stack access methods.

According to a sixth aspect of the invention, the architecturally defined stack access methods include memory accesses that use one of a stack pointer, a frame pointer, and an argument pointer.

According to a seventh aspect of the invention, the architecturally defined stack access methods include push, pop, and other stack manipulation operations.

According to an eighth aspect of the invention, there is provided a method for renaming memory references to stack locations in a computer processing system. The method includes the step of determining whether a load instruction references a location in a local stack using an

architecturally defined register for accessing a stack location. It is determined whether a rename register exists for the referenced location in the local stack, when the load instruction references the location using the
5 architecturally defined register. The reference to the location is replaced by a reference to the rename register, when the rename register exists.

These and other aspects, features and advantages of the present invention will become apparent from the
10 following detailed description of preferred embodiments, which is to be read in connection with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

15 FIG. 1 is a diagram illustrating a typical memory layout of a computer program according to the prior art;

FIG. 2 is a flow diagram illustrating a method for performing the initial processing of a single instruction according to an illustrative embodiment of the present
20 invention;

FIG. 3 is a flow diagram illustrating the processing of load operations according to an illustrative embodiment of the present invention;

FIG. 4 is a flow diagram illustrating the processing of a load operation according to another illustrative embodiment of the present invention;

FIG. 5 is a flow diagram illustrating the processing of store operations according to an illustrative embodiment of the present invention; and

FIG. 6 is a block diagram illustrating a superscalar out-of-order processor to which the present invention is applied.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

The present invention is directed to methods for renaming stack references in a computer processing system. According to the invention, the stack references are renamed to processor-internal registers. By concentrating on the frequent rename opportunities for stack references, the renaming architecture can be more efficiently implemented than prior art approaches to renaming memory references.

Such efficiency is derived from two aspects. With respect to the first aspect, the names are easier to determine, since references to the processor stack use a limited number of addressing registers such as the stack and frame pointers. This reduces the possible ambiguities that

can arise in the renaming of memory locations using
different general purpose registers. While accesses using
other registers are possible, they are sufficiently
infrequent to be handled using simple disambiguation
5 techniques.

With respect to the second aspect, the need to perform
consistency-preserving operations in a multiprocessor system
is significantly reduced, since stack references are usually
referenced only from the local process. Again, references
10 from other processors are possible, but since they are
infrequent, they can be resolved using simple techniques.

It is to be appreciated that by renaming stack
references to processor registers, stack references become
amenable to a number of optimizations typically applied to
15 processor registers, such as reordering of references,
renaming to resolve anti-dependencies, and speculative
execution of write operations.

To facilitate a clear understanding of the present
invention, definitions of terms employed herein will now be
20 given. A load instruction refers to any instruction
performing a memory read-access and (optionally)
computations based on the loaded value. Thus, a load
instruction may include, for example, logic, arithmetic and

other instructions which employ data from memory locations as operands. A store instruction refers to any instruction performing a memory write-access and, optionally, computations. Out-of-order execution is a technique by which the operations in a sequential stream of instructions are reordered so that operations appearing later are executed earlier, if the resources required by the later appearing operations are free. Thus, an out-of-order instruction may be created, either statically or dynamically, by moving an instruction from its original position in a sequence of instructions to an earlier position in the sequence of instructions.

The following description and corresponding examples will be given based on two instructions (unless otherwise noted), a first instruction which is executed out-of-order before a second, logically preceding instruction, which will be termed an in-order instruction. Thus, unless otherwise noted, the designation 'in-order' refers only to the sequential relationship between the logically preceding in-order instruction and the first 'out-of-order' instruction. It is to be noted that the above two instructions (i.e., both the in-order and the out-of-order

load instructions) may be in-order or out-of-order with respect to a third instruction (and so forth).

A general description of the present invention will now be provided to introduce the reader to the concepts of the invention. Subsequently, more detailed descriptions of various aspects of the invention will be provided.

The present invention simplifies the issue of memory renaming by renaming only memory stack references (as opposed to renaming all memory references). Such an approach is advantageous for at least the following reasons:

1. Performance degradations due to the serializing effects of stack references are artificially introduced by the translation process and not intrinsic in the program.
2. Significant performance gain can be achieved by simply renaming the stack references.
3. The "names" of stack accesses are easy to determine since the stack and frame pointers have well-defined semantics (unlike other registers which could point to any location in memory).
4. Since stack references are usually local to one process and a single processor, and are made through the stack or frame pointers, the importance of

multiprocessor consistency is reduced.

5. Detecting naming equivalence is easy because references to the stack are usually only through the stack or frame pointers (the frame pointer is usually at a well-defined distance from the stack pointer).

Consistency between different references to memory is resolved by observing that stack references are usually only to the local processor stack and only through the registers specifically designated to manage the stack, such as the stack pointer (SP), the frame pointer (FP), and an optional argument pointer (AP).

References to the stack area of one processor by any means other than these specifically designated stack-management registers result in the performing of actions to preserve consistency. Such references can be easily detected. For example, accesses to the local processor stack by a general register (a register other than a stack register such as, for example, SP, FP, and AP) can be detected by tagging page table entries that contain the stack, and indicating the condition in the memory unit. The processor can then handle this situation in hardware or raise an exception and resolve the condition in the

exception handler. As another example, in a multiprocessing system, accesses to the stack of another processor can be detected by not making entries available in the translation lookaside buffer (TLB), thereby taking an exception and
5 handling the condition in software, or by indicating in the TLB that a page is used to store the stack on a remote processor.

FIG. 2 is a flow diagram illustrating a method for performing the initial processing of a single instruction according to an illustrative embodiment of the present
10 invention.

The instruction is fetched from the program memory 100 (step 210). It is then determined whether the fetched instruction is a memory operation (i.e., whether the fetched
15 instruction references memory 100) (step 212). If so, then the memory operation is processed in accordance with the present invention (step 214). The particular processing steps are shown in detail in FIGs. 3 and 4 for an instruction referencing the memory for read access, and in
20 FIG. 5 for an instruction referencing the memory for write access.

If the fetched instruction is not a memory operation, then references to processor registers are renamed (step

216). It is to be appreciated that step 216 is optional and, thus, may be omitted if so desired. The instruction is then entered in the dispatch table for dispatch to one of the execution units (step 218), and the method is terminated.

FIG. 3 is a flow diagram illustrating the processing of a load operation (e.g., step 214 of FIG. 2) according to an illustrative embodiment of the present invention. In general, if a rename register is found for a memory read reference, then the memory read reference is replaced by a reference to the rename register. Otherwise, a load from the stack location stored in memory is performed.

According to the method of FIG. 3, it is determined whether the load instruction references a location in the local stack using the stack pointer SP or frame pointer FP (step 310). With respect to some architectures, step 310 could optionally determine whether the load instruction references any additional pointers used to access a stack location, such as the argument pointer AP in the DEC VAX architecture. The load instruction does not have to reference the SP, FP or AP registers explicitly, but can be an instruction such as push or pop (e.g., as found in the

Intel x86 architecture) which references at least one such pointer implicitly.

If the load instruction references a location in the local stack using the stack pointer SP or frame pointer FP,

5 then the method proceeds to step 320. Otherwise, it is determined whether the load instruction references a location in a stack using any other register (step 314).

Unlike the determination made in step 310, which was concerned with only a local stack location, the

10 determination made in step 314 encompasses both a local or a remote stack location.

Step 314 can be performed during either the decode, address generation, or memory access phase. According to an illustrative embodiment of the present invention, one
15 implementation of step 314 may include marking TLB entries of pages in stack 110 (see FIG. 1) as containing stack references.

If the load instruction does not reference a location in a stack using any other register, a normal load operation
20 (i.e., a load operation from main memory or a cache) is performed and then the method is terminated (step 316). However, if the load instruction does reference a location in a stack using any other register, then a

consistency-preserving mechanism is executed to perform a load operation from the stack area (step 318). The consistency preserving mechanism can be implemented in hardware, software, or a combination thereof. Illustrative
5 embodiments of the consistency preserving mechanism are described in detail hereinbelow.

At step 320, it is determined whether a rename register exists for the referenced location in the local stack. Step 320 can be performed using either a symbolic
10 address, i.e., "(SP)+100" for a memory reference with displacement 100 from the stack pointer, or by actually computing the address of the referenced location.

If a rename register exists for the referenced location in the local stack, then the reference to the stack
15 location is replaced by a reference to the rename register and the method is terminated (step 322). However, if a rename register does not exist for the referenced location in the local stack, then a normal load instruction (i.e., a load instruction from main memory or a cache) is inserted in
20 the instruction stream and the method is terminated (step 324).

Restricting memory renaming to stack references provides advantages that simplify address resolution with

respect to the above prior art approaches for memory renaming. Thus, unlike data dependency prediction as used by A. Moshovos and G. Sohi, in the above referenced article entitled "Streamlining Inter-operation Memory Communication via Data Dependence Prediction", actual dependencies can be
5 determined by decoding the instruction.

Moreover, unlike the approach for other symbolic renaming techniques, symbolic renaming of stack references does not require complex logic to determine equivalencies, such as that described in U.S. Patent No. 5,838,941,
10 entitled "Out-of-order Superscalar Microprocessor with a Renaming Device that Maps Instructions from Memory to Registers", issued on November 17, 1998, the disclosure of which is incorporated herein by reference. With respect to
15 the approach of the present invention, all references are through one of the stack registers which are at an easily determinable offset from each other.

It is to be appreciated that the method of FIG. 3 requires adjustment of references as the stack pointer value is changed (e.g., on subroutine calls). However, such
20 adjustments may be readily made by one of ordinary skill in the related art.

Using absolute addresses is also simplified, since the values of the stack pointer and frame pointer are easy to determine and change infrequently. In particular, changes to these registers involve simple arithmetic, which can be performed speculatively with a shadowed stack pointer used only for renaming in the stack reference rename logic.

FIG. 4 is a flow diagram illustrating the processing of a load operation (e.g., step 214 in FIG. 2) according to another illustrative embodiment of the present invention. In general, if a rename register is found for a stack location, then the memory reference is replaced by a reference to the rename register. Otherwise, a new rename register is allocated, an instruction to load the value from the stack to the rename register is inserted into the instruction stream, and the original reference to the memory location is replaced by a reference to the newly allocated rename register.

According to the method of FIG. 4, it is determined whether the load instruction references a location in the local stack using the stack pointer SP or frame pointer FP (step 410). With respect to some architectures, step 410 could optionally determine whether the load instruction references any additional pointers used to access a stack

location, such as the argument pointer AP in the DEC VAX architecture. The load instruction does not have to reference the SP, FP or AP registers explicitly, but can be an instruction such as push or pop (e.g., as found in the Intel x86 architecture) which references at least one such pointer implicitly.

If the load instruction references a location in the local stack using the stack pointer SP or frame pointer FP, then the method proceeds to step 420. Otherwise, it is determined whether the load instruction references a location in a stack using any other register (step 414). Step 414 can be performed during either the decode, address generation, or memory access phase. According to an illustrative embodiment of the present invention, one implementation of step 414 may include marking TLB entries of pages in the stack memory area (see FIG. 1) as containing stack references.

If the load instruction does not reference a location in a stack using any other register, then a normal load operation (i.e., a load operation from main memory or a cache) is performed and the method is terminated (step 416). However, if the load instruction does reference a location in a stack using any other register, then a

consistency-preserving mechanism is executed to perform a load operation from the stack area (step 418). The consistency preserving mechanism can be implemented in hardware, software, or a combination thereof. Illustrative
5 embodiments of the consistency preserving mechanism are described in detail hereinbelow.

At step 420, it is determined whether a rename register exists for the referenced location. This test can be performed using either a symbolic address, i.e.,
10 "(SP)+100" for a memory reference with displacement 100 from the stack pointer, or by actually computing the address of the referenced location. Restricting memory renaming to stack references provided advantages as described above with respect to FIG. 3.

15 Irrespective of the naming technique used, if a rename register exists for the referenced location, then the reference to the stack location is replaced by a reference to the rename register and the method is terminated (step 422). However, if a rename register does not exist for the
20 referenced location, then a rename register is allocated for the stack location referenced by the load operation (step 424). Then, a load instruction is inserted in the instruction stream to load the value from the processor into

the newly allocated rename register (step 426), and the method returns to step 422.

FIG. 5 is a flow diagram illustrating the processing of a store operation (e.g., in unit 620 of FIG. 6) according to an illustrative embodiment of the present invention.

It is determined whether the store instruction references a location in the local stack using the stack pointer SP or frame pointer FP (step 510). With respect to some architectures, step 510 could optionally determine whether the store instruction references any additional pointers used to access a stack location, such as the argument pointer AP in the DEC VAX architecture. The store instruction does not have to reference the SP, FP or AP registers explicitly, but can be an instruction such as push or pop (e.g., as found in the Intel x86 architecture) which references at least one such pointer implicitly.

If the store instruction references a location in the local stack using the stack pointer SP or frame pointer FP, then the method proceeds to step 520. Otherwise, it is determined whether the store instruction references a location in a stack using any other register (step 514). Unlike the determination made in step 510 which was concerned with only a local stack location, the

determination made in step 514 encompasses both a local or a remote stack location. Step 514 can be performed during either the decode, address generation, or memory access phase. According to an illustrative embodiment of the present invention, one implementation of step 514 may include marking TLB entries of pages in the stack memory area (see FIG. 1) as containing stack references.

If the store instruction does not reference a location in a stack using any other register, then a normal store operation (i.e., a store operation from main memory or a cache) is performed and the method is terminated (step 516). However, if the instruction does reference a location in a stack using any other register, then a consistency-preserving mechanism is executed to perform a store operation to the stack area (step 518). The consistency preserving mechanism can be implemented in hardware, software, or a combination thereof. Illustrative embodiments of the consistency preserving mechanism are described in detail hereinbelow.

At step 520, a new rename register is allocated for the stack location referenced by the store operation. The rename register can be named using either a symbolic address, i.e., "(SP)+100" for a memory reference with

displacement 100 from the stack pointer, or by actually computing the address. Restricting memory renaming to stack references provides advantages as described above with respect to FIG. 3.

5 The memory reference is replaced by a reference to the newly allocated rename register (step 522). An instruction to store the value from the rename register to memory is inserted into the instruction stream (step 524), and the method is terminated. It is to be appreciated that step 524
10 is optional and, thus, may be omitted if so desired.

A conventional implementation of a processor capable of dynamically scheduling instructions (an out-of-order issue processor) includes the following features:

1. A mechanism for issuing instructions out-of-order,
15 which includes the ability to detect dependencies among the instructions, rename the registers used by an instruction, and detect the availability of the resources used by an instruction;
2. A mechanism for maintaining the out-of-order state
20 of the processor, which reflects the effects of instructions as they are executed (out-of-order);
3. A mechanism for retiring instructions in program

order, and simultaneously updating the in-order state with the effects of the instructions being retired; and

4. A mechanism for retiring an instruction in program order without updating the in-order state (effectively canceling the effects of the instruction being retired), and for resuming in-order execution of the program starting at the instruction being retired (which implies canceling all the effects present in the out-of-order state).

Mechanism 3 from the list above is used to retire instructions when the effects of the instructions being retired are correct. Alternatively, mechanism 4 is used whenever there is some abnormal condition resulting from the execution of the instruction being retired or from some external event.

FIG. 6 is a functional block diagram of a conventional computer processing system (e.g., including a superscalar processor) to which the present invention may be applied. The system of FIG. 6 supports reordering of memory operations using the mechanisms listed above, but excluding the ability to rename and execute references to the program stack out-of-order. The system consists of: a memory

subsystem 601; a data cache 602; an instruction cache 604;
and a processor unit 600. The processor unit 600 includes:
an instruction queue 603; several memory units (MUs) 605 for
performing load and store operations; several functional
5 units (FUs) 607 for performing integer, logic and
floating-point operations; a branch unit (BU) 609; a
register file 611; a register map table 620; a
free-registers queue 622; a dispatch table 624; a retirement
queue 626; and an in-order map table 628. This exemplary
10 organization is based on the one described in the article by
M. Moudgill, K. Pingali, and S. Vassiliadis, "Register
Renaming and Dynamic Speculation: An Alternative Approach",
Proceedings of the 26th Annual International Symposium on
Microarchitecture, pp. 202-13 (December 1993).

15 In the processor depicted in FIG. 6, instructions are
fetched from instruction cache 604 (or from memory subsystem
601, when the instructions are not in instruction cache 604)
under the control of branch unit 609, placed in instruction
queue 603, and subsequently dispatched from instruction
20 queue 603. The register names used by the instructions for
specifying operands are renamed according to the contents of
register map table 620, which specifies the current mapping
from architected register names to physical registers. The

architected register names used by the instructions for specifying the destinations for the results are assigned physical registers extracted from free-registers queue 622, which contains the names of physical registers not currently being used by the processor. The register map table 620 is updated with the assignments of physical registers to the architected destination register names specified by the instructions. Instructions with all their registers renamed are placed in dispatch table 624. Instructions are also placed in retirement queue 626, in program order, including their addresses, and their physical and architected register names. Instructions are dispatched from dispatch table 624 when all the resources to be used by such instructions are available (physical registers have been assigned the expected operands, and functional units are free). The operands used by the instruction are read from register file 611, which typically includes general-purpose registers (GPRs), floating-point registers (FPRs), and condition registers (CRs). Instructions are executed, potentially out-of-order, in a corresponding memory unit 605, functional unit 607 or branch unit 609. Upon completion of execution, the results from the instructions are placed in register file 611. Instructions in dispatch table 624 waiting for

the physical registers set by the instructions completing execution are notified. The retirement queue 626 is notified of the instructions completing execution, including whether they raised any exceptions. Completed instructions are removed from retirement queue 626, in program order (from the head of the queue). At retirement time, if no exceptions were raised by an instruction, then in-order map table 628 is updated so that architected register names point to the physical registers in register file 611 containing the results from the instruction being retired; the previous register names from in-order map table 628 are returned to free-registers queue 622.

On the other hand, if an instruction has raised an exception, then program control is set to the address of the instruction being retired from retirement queue 626. Moreover, retirement queue 626 is cleared (flushed), thus canceling all unretired instructions. Further, the register map table 620 is set to the contents of in-order map table 628, and any register not in in-order map table 628 is added to free-registers queue 622.

In addition to the components above, superscalar processors may contain other components such as branch-history tables to predict the outcome of branches.

According to the present invention, a conventional superscalar processor that supports reordering of load instructions with respect to preceding load instructions (as shown in FIG. 6) may be augmented with the following:

- 5 A. A first mechanism for detecting load and store operations (explicit load or store operations, or any other operations which reference memory using the stack or frame pointers, as typically found in CISC instruction sets).
- 10 B. A second mechanism for converting a load or store operation into a reference to an internal register.
- C. A third mechanism to detect whether any other load instruction (which does not address memory through the stack and frame pointers) refers to the
15 processor stack, and providing the value to the instruction.
- D. A fourth mechanism for detecting a memory reference to another processor's stack area, and retrieve the value.

20 In addition, mechanism 4 above is preferably extended to force all stack references currently stored in rename registers to processor memory. This is particularly simple if, in step 524 of FIG. 5, memory write instructions are

entered into the instruction stream to record all stack references in processor memory in-order.

The mechanisms provided by this invention are used in conjunction with the mechanisms available in the conventional out-of-order processor depicted in FIG. 6, as follows. The first mechanism, which detects memory operations, is implemented in the register map table 620. If the memory operation refers to a stack reference, it is converted into a reference to a rename register using the second mechanism. Rename registers for stack references may be shared with those used for general purpose registers, or they may be implemented as distinct processor resources.

The third and fourth mechanisms are implemented in memory units 605 and in system software following a hybrid organization.

The third mechanism for the detection of interference between general-register based memory accesses and local stack references is preferably performed by marking TLB entries referring to the processor stack. In this embodiment, references to such pages using a general register cause the processor to discard all speculative state and perform the operation in-order. According to this embodiment, the in-order values of stack references have

been stored to the processor memory by store instructions which were inserted in program order (step 524 of FIG. 5). Processing then continues with the instruction following the current instruction.

5 The fourth mechanism is implemented by not loading page translations for memory locations which are used as program stack in remote processors. As a result, a reference to a stack location on a remote processor raises an exception, whereupon the system software can synchronize
10 the memory read request from the remote processor's stack area with that processor.

 According to one illustrative embodiment of the present invention, this is performed in software by sending a request to the operating system kernel executing on the
15 remote processor. According to another illustrative embodiment of the present invention, this request and the necessary actions to ensure consistency are performed in hardware.

 A brief description of the consistency preserving
20 mechanism of steps 318, 418, and 518 of FIGs. 3, 4, and 5, respectively, will now be given. In-order store operations may be performed (at optional step 524 of FIG. 5) by performing a load operation from the corresponding address.

If in-order store operations are not performed for stack references, then one implementation of a consistency-preserving mechanism forces all stack rename registers to its corresponding stack locations. This can be implemented as either a distinct mechanism, or by activating mechanism 4 of FIG. 6 which discards all out-of-order operations and has been extended to preferably for all stack rename locations to memory. In another embodiment, load references to the local processor stack may also be bypassed directly from the rename registers.

If multiple values are present, then the last value written before the current load operation must be retrieved in steps 318 and 418 of FIGs. 3 and 4, respectively. One way to do this is to discard all instructions following the present load operation.

Consistency-preserving mechanisms for store operations can include discarding all out-of-order state by activating mechanism 4 of FIG. Mechanism 4 discards all out-of-order operations and can be preferably extended to all stack rename locations to memory. When execution restarts, the modified value will loaded from memory by step 316 of FIG. 3 or step 426 of FIG. 4.

Special care needs to be taken if stack load and store operations refer to data types of different sizes. If the load operation refers to a subset of data bits provided by the store operation, then these can be extracted in a simple manner. If the data referred to by a load operation is resident in one or more stack rename registers and possibly processor main memory, then more complex data gathering is required.

According to one embodiment of the present invention, at least all overlapping registers are written to main memory and a memory read operation is performed. This is particularly simple if, in step 524 of FIG. 5, memory write instructions are entered into the instruction stream to record all stack references in processor memory in-order.

According to another embodiment of the present invention, the processor discards all speculative state, forces all stack references to memory, and performs the memory load operation in-order. Again, this is particularly simple if, in step 524 of FIG. 5, memory write instructions are entered into the instruction stream to record all stack references in processor memory in-order.

When the processor rename mechanism runs out of rename registers, rename registers which have no references in

pending instructions may be reclaimed and their contents written to the processor memory. This may require that the value stored by the rename register be written to memory if no in-order stores have been inserted in step 524 of FIG 5.

5 In an optimized embodiment, a predictor is used to decide which stack references to rename into processor registers in step 424 of FIG. 4 and step 520 of FIG. 5 to reduce the number of registers allocated to stack references. Stack references not allocated in a processor
10 register are performed by using load and store instructions into the main memory.

According to one embodiment of the invention, rename registers can be shared for processor register and stack reference renaming. According to another embodiment of the
15 invention, they are separate resources.

According to an optimized embodiment, renaming of stack references is combined with the elimination of copy operations, thereby reducing the critical path.

According to one embodiment, stack references are
20 named using symbolic names of the form "stack pointer value + displacement". In this case, synchronization of names is required when the stack pointer contents are changed. Also, the distance between frame pointer and stack pointer is used

to translate FP-relative references to SP-relative references.

According to another embodiment, addresses (e.g., effective, virtual or physical) are used to name the stack pointer. In an optimized embodiment, the stack pointer (and optionally, frame pointer, argument pointer, etc.) are shadowed in the rename unit, and speculatively adjusted in synchronization with the instruction stream being fetched. This reduces the number of memory ports necessary to the register file and allows more aggressive speculation during renaming.

According to one embodiment, consistency-preserving operations for accesses to the local stack are implemented in hardware by associating each renamed stack reference with address information and performing a lookup of renamed stack references to determine whether the requested data is in a rename register or in an actual memory location. According to another embodiment, this is implemented using software only: a reference to a stack area using a general register causes an exception, and software synchronizes the renamed values with the present request.

Although the illustrative embodiments have been described herein with reference to the accompanying

drawings, it is to be understood that the present system and method is not limited to those precise embodiments, and that various other changes and modifications may be affected therein by one skilled in the art without departing from the scope or spirit of the invention. All such changes and modifications are intended to be included within the scope of the invention as defined by the appended claims.

5

WHAT IS CLAIMED IS:

1. A method for renaming memory references to stack locations in a computer processing system, comprising the steps of:

detecting stack references that use architecturally defined stack access methods; and

replacing the stack references with references to processor-internal registers.

2. The method according to claim 1, further comprising the step of synchronizing an architected state between the processor-internal registers and a main memory of the computer processing system.

3. The method according to claim 2, wherein said synchronizing step comprises the step of inserting in-order write operations for all of the stack references that are write stack references.

4. The method according to claim 1, further comprising the step of performing a consistency-preserving

operation for a stack reference that does not use the architecturally defined stack access methods.

5 5. The method according to claim 4, wherein said step of performing a consistency-preserving operation comprises the step of bypassing a value from a given processor-internal register to a load operation that references a stack area and that does not use the architecturally defined stack access methods.

10 6. The method according to claim 4, further comprising the step of synchronizing an architected state between the processor-internal registers and a main memory of the computer processing system, and wherein said step of performing a consistency-preserving operation comprises the
15 step of recovering an in-order value for the stack reference from the main memory, upon performing said synchronizing step.

20 7. The method according claim 6, wherein the in-order value is written to the main memory by an in-order write operation inserted into an instruction stream containing an

instruction corresponding to the stack reference, when the stack reference is a write stack reference.

5 8. The method according to claim 6, further comprising the step of writing the in-order value to the main memory in response to a load operation that does not use the architecturally defined stack access methods.

10 9. The method according to claim 4, wherein said step of performing a consistency-preserving operation comprises the steps of:

discarding all out-of-order state of a processor the system;

15 synchronizing an architected state between the processor-internal registers and a main memory of the computer processing system; and

restarting execution after a store operation has been performed that does not use the architecturally defined stack access methods.

20

10. The method according to claim 1, wherein the architecturally defined stack access methods comprise memory

accesses that use at least one of a stack pointer, a frame pointer, and an argument pointer.

11. The method according to claim 1, wherein the
5 architecturally defined stack access methods comprise push, pop, and other stack manipulation operations.

12. A method for renaming memory references to stack
10 locations in a computer processing system, comprising the steps of:

determining whether a load instruction references a
location in a local stack using an architecturally defined
register for accessing a stack location;

15 determining whether a rename register exists for the
referenced location in the local stack, when the load
instruction references the location using the
architecturally defined register; and

replacing the reference to the location by a reference
to the rename register, when the rename register exists.

20 13. The method according to claim 12, wherein the
architecturally defined register corresponds to a pointer
for accessing stack locations.

14. The method according to claim 13, wherein the pointer for accessing the stack locations is one of a stack pointer, a frame pointer, and an argument pointer.

5

15. The method according to claim 12, wherein the architecturally defined stack access methods comprise push, pop, and other stack manipulation operations.

10

16. The method according to claim 12, wherein said step of determining whether the renaming register exists comprises the step of computing one of a symbolic address and an actual address of the location.

15

17. The method according to claim 12, wherein said step of determining whether the rename register exists is performed during one of a decode, an address generation, and a memory access phase.

20

18. The method according to claim 12, further comprising the step of performing the load instruction from one of a main memory and a cache of the system, when the rename register does not exist.

19. The method according to claim 12, further comprising the step of determining whether the load instruction references a location in any stack, including the local stack, using another register, when the load instruction does not reference the location using the architecturally defined register.

20. The method according to claim 19, wherein said step of determining whether the load instruction references the location using the other register comprises the step of marking translation lookaside buffer (TLB) entries of pages in the local stack as containing stack references, when the load instruction references the location using the other register.

21. The method according to claim 19, further comprising the step of performing the load instruction from one of a main memory and a cache of the system, when the load instruction does not reference the location using the other register.

22. The method according to claim 19, further comprising the step of executing a consistency-preserving mechanism to perform the load instruction from the stack area, when the load instruction references the location using the other register.

23. The method according to claim 12, further comprising the step of allocating a rename register for the location, when the rename register does not exist.

24. The method according to claim 23, further comprising the step of inserting an operation, into an instruction stream containing the load instruction, to load the location from a processor of the system to the allocated rename register, upon allocating the rename register.

25. The method according to claim 24, further comprising the step of replacing the reference to the location by a reference to the allocated rename register, upon inserting the operation.

26. A method for renaming memory references to stack locations in a computer processing system, comprising the steps of:

determining whether a store instruction references a location in a local stack using an architecturally defined register for accessing a stack location;

allocating a rename register for the location, when the store instruction references the location using the architecturally defined register; and

replacing the reference to the location by a reference to the rename register.

27. The method according to claim 26, further comprising the step of inserting an operation, into an instruction stream containing the store instruction, to store the location from the rename register to a main memory of the system, upon replacing the reference to the location by the reference to the rename register.

28. The method according to claim 26, further comprising the step of determining whether the store instruction references a location in any stack, including the local stack, using another register, when the store

instruction does not reference the location using the architecturally defined register.

29. The method according to claim 28, further comprising the step of performing the store instruction from one of a main memory and a cache of the system, when the store instruction does not reference the location using the other register.

30. The method according to claim 28, further comprising the step of executing a consistency-preserving mechanism to perform the store instruction to the stack area, when the store instruction references the location using the other register.

31. A program storage device readable by machine, tangibly embodying a program of instructions executable by the machine to perform method steps for renaming memory references to stack locations in a computer processing system, the method steps comprising:

detecting stack references that use architecturally defined stack access methods; and

replacing the stack references with references to processor-internal registers.

32. The program storage device according to claim 31,
5 further comprising the step of synchronizing an architected state between the processor-internal registers and a main memory of the computer processing system.

33. The program storage device according to claim 32,
10 wherein said synchronizing step comprises the step of inserting in-order write operations for all of the stack references that are write stack references.

34. The program storage device program storage device
15 according to claim 31, further comprising the step of performing a consistency-preserving operation for a stack reference that does not use the architecturally defined stack access methods.

20 35. The program storage device according to claim 34, wherein said step of performing a consistency-preserving operation comprises the step of bypassing a value from a given processor-internal register to a load operation that

references a stack area and that does not use the architecturally defined stack access methods.

36. The program storage device according to claim 34,
5 further comprising the step of synchronizing an architected state between the processor-internal registers and a main memory of the computer processing system, and wherein said step of performing a consistency-preserving operation
10 comprises the step of recovering an in-order value for the stack reference from the main memory, upon performing said synchronizing step.

37. The program storage device according claim 36,
15 wherein the in-order value is written to the main memory by an in-order write operation inserted into an instruction stream containing an instruction corresponding to the stack reference, when the stack reference is a write stack reference.

20 38. The program storage device according to claim 36, further comprising the step of writing the in-order value to the main memory in response to a load operation that does not use the architecturally defined stack access methods.

39. The program storage device according to claim 34,
wherein said step of performing a consistency-preserving
operation comprises the steps of:

5 discarding all out-of-order state of a processor the
system;

 synchronizing an architected state between the
processor-internal registers and a main memory of the
computer processing system; and

10 restarting execution after a store operation has been
performed that does not use the architecturally defined
stack access methods.

METHODS FOR RENAMING STACK REFERENCES
IN A COMPUTER PROCESSING SYSTEM

ABSTRACT OF THE INVENTION

5 According to one aspect of the invention, there is
provided a method for renaming memory references to stack
locations in a computer processing system. The method
includes the steps of detecting stack references that use
architecturally defined stack access methods, and replacing
10 the stack references with references to processor-internal
registers. The architecturally defined stack access methods
include memory accesses that use one of a stack pointer, a
frame pointer, and an argument pointer. Moreover, the
architecturally defined stack access methods include push,
15 pop, and other stack manipulation operations.

1/6
Michael R. Gschwind
40999-357 (JPS) 8728-320

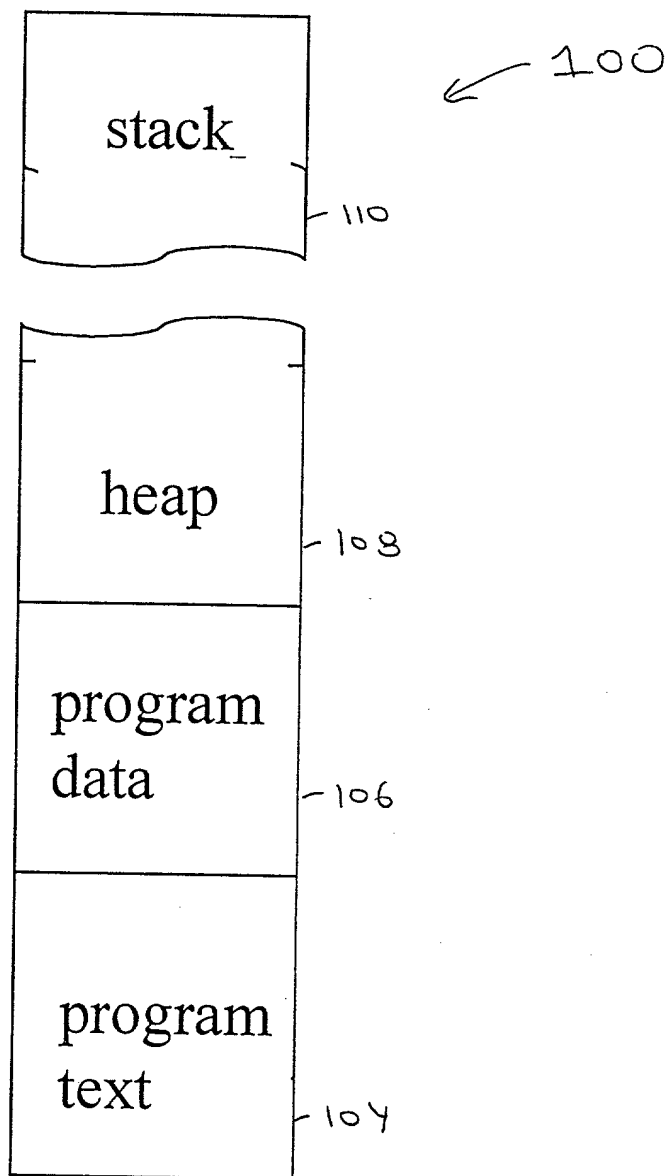


FIG. 1

2/6
10999-357 (8728-320)

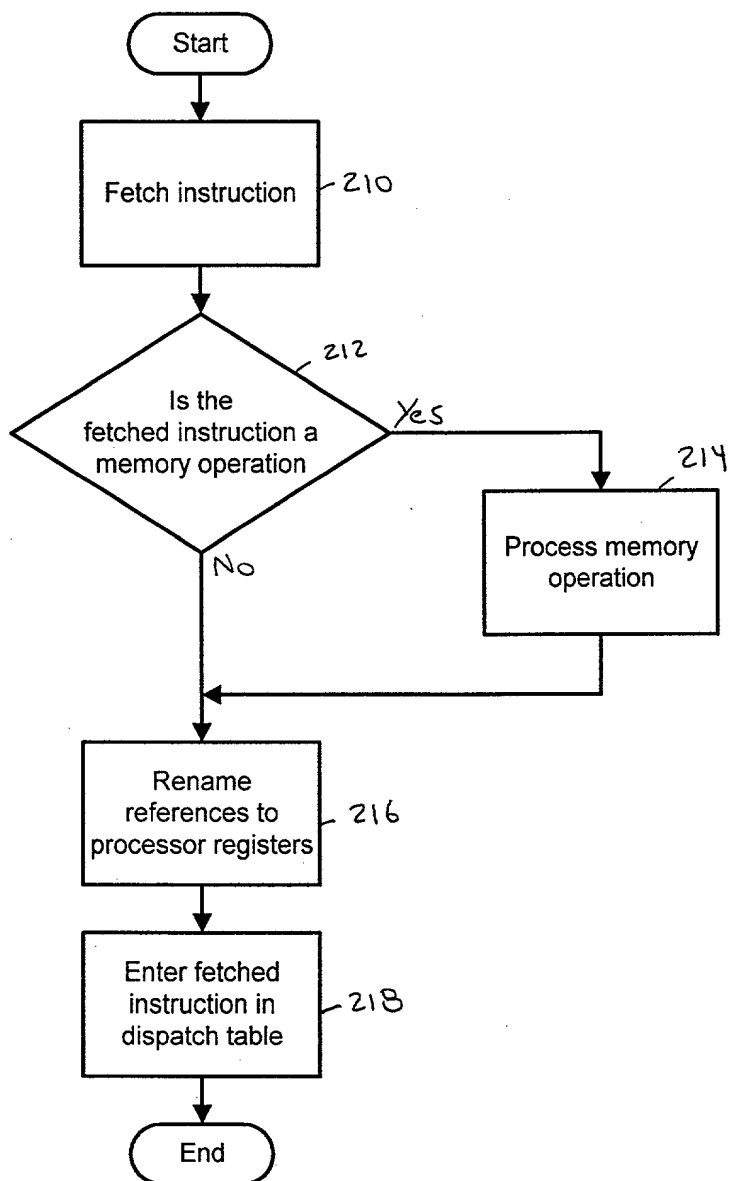


FIG. 2

3/6

40999-357 (8728-320)

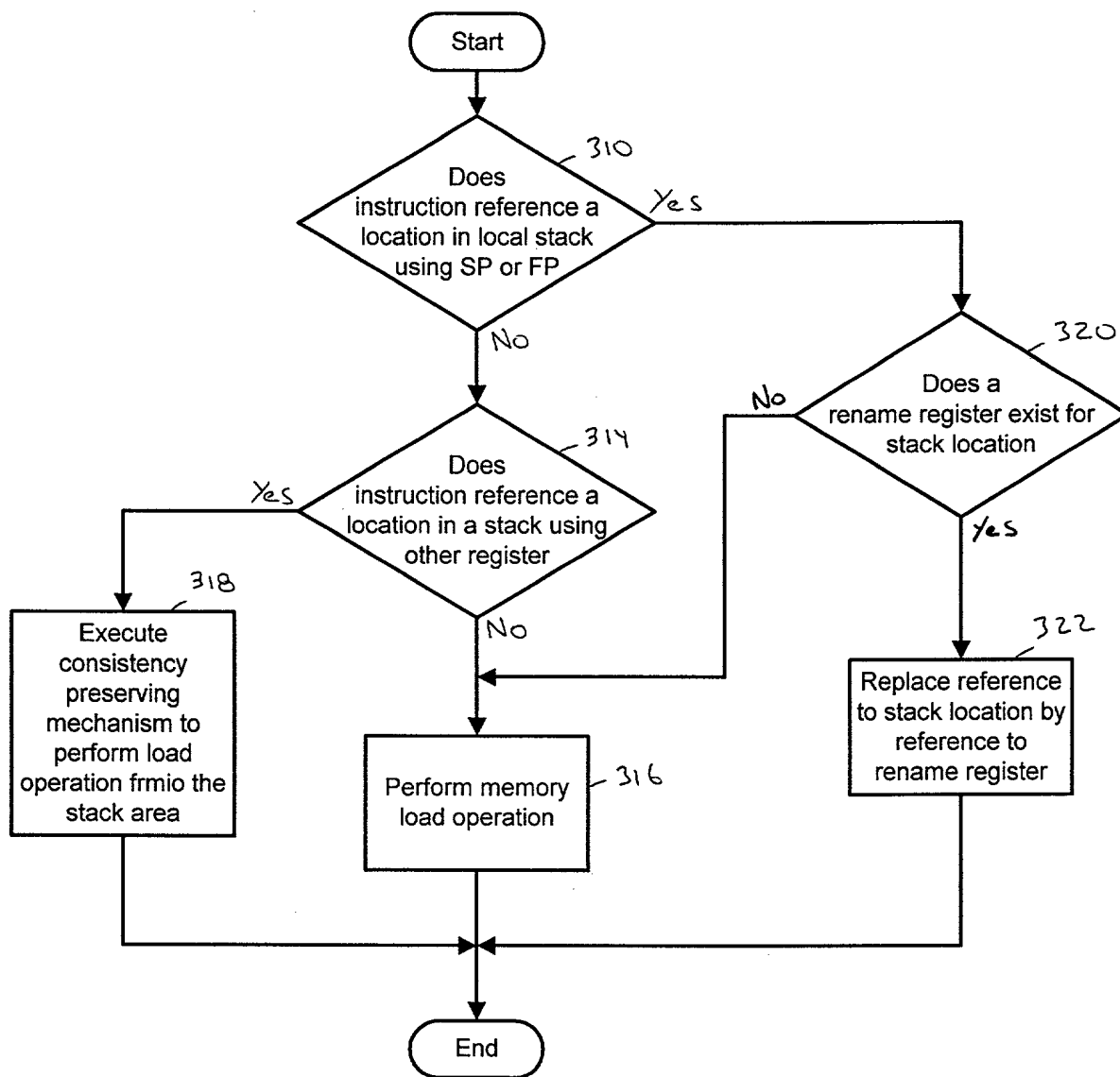


FIG. 3

4/6
Y0999-357 (8728-320)

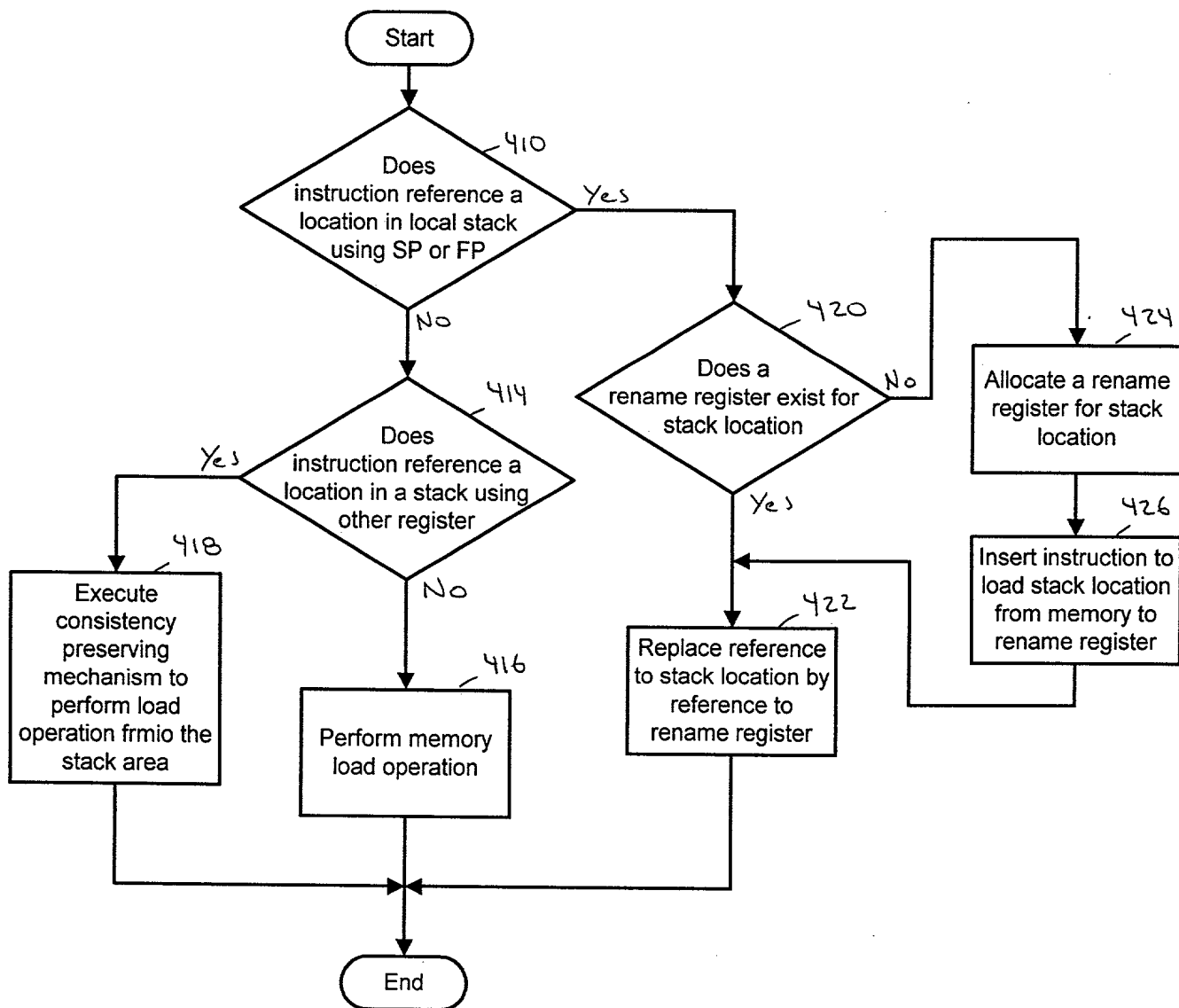


FIG. 4

5/6
10999-357 (8728-320)

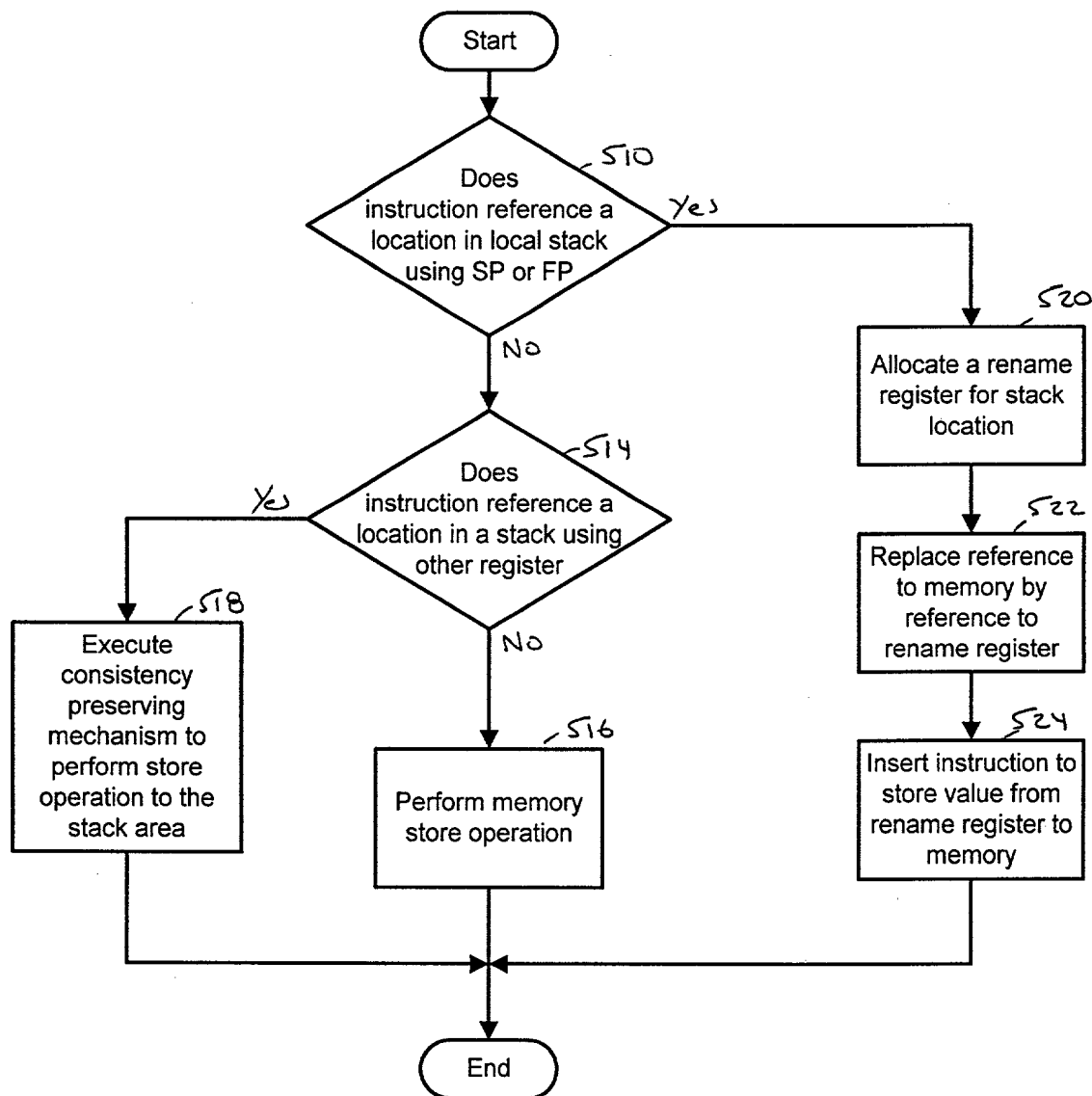


FIG. 5

6/6
10999-357(8728-320)

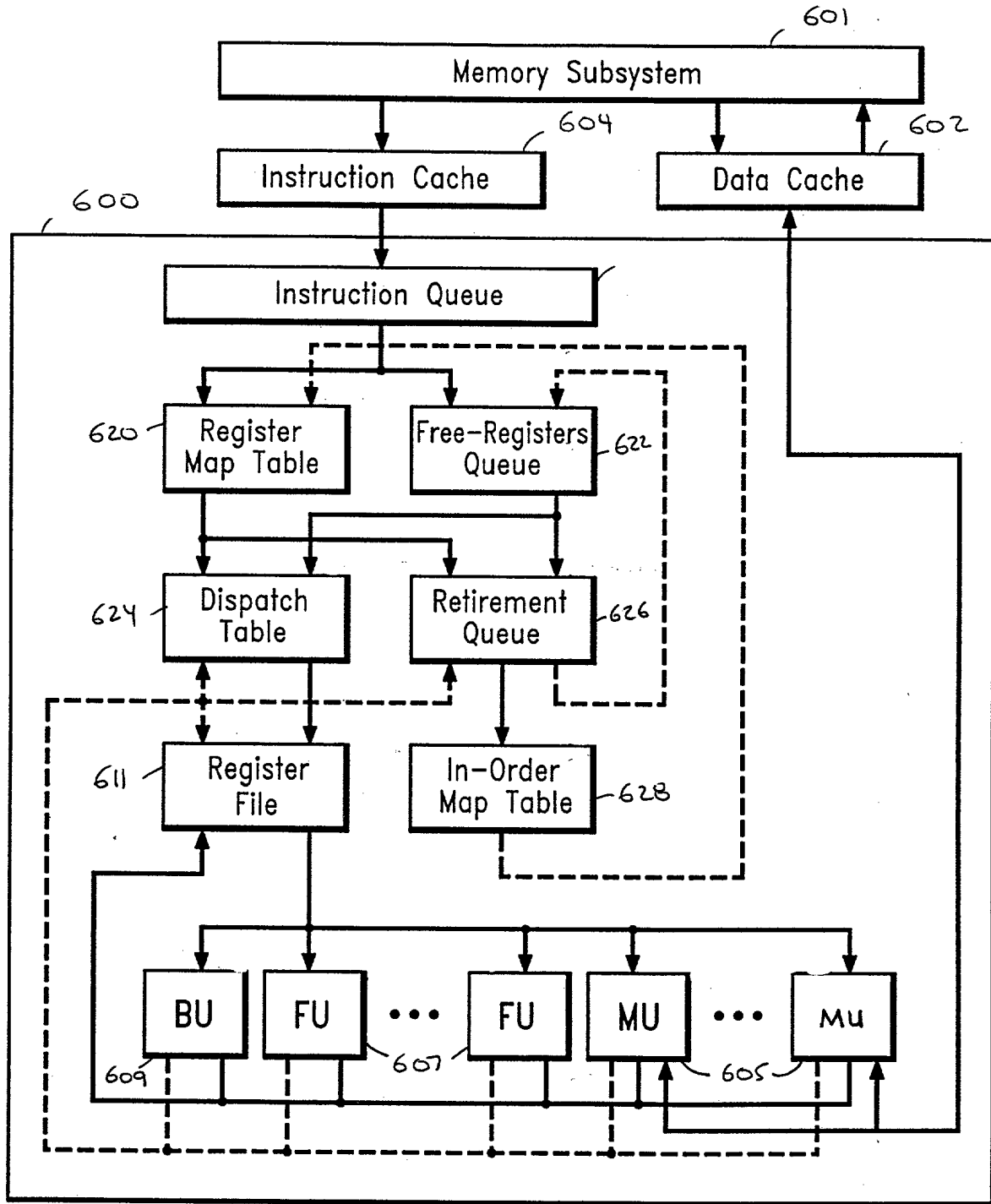


FIG. 6

AS A BELOW NAMED INVENTOR, I hereby declare that:
My residence, post office address and citizenship are as stated next to my name.

I believe that I am the original, first and sole (if only one name is listed below), or an original, first and joint inventor (if plural names are listed below), of the subject matter which is claimed and for which a patent is sought on the invention entitled:

TITLE: METHODS FOR RENAMING STACK REFERENCES IN A COMPUTER PROCESSING SYSTEM

the specification of which either is attached hereto or indicates an attorney docket no. YO999-357 (8728-320), or:

☐ was filed in the U.S. Patent & Trademark Office on _____ and assigned Serial No. _____,
☐ and (if applicable) was amended on _____

I hereby state that I have reviewed and understand the contents of the above-identified specification, including the claims, as amended by any amendment referred to above. I acknowledge the duty to disclose information which is material to patentability and to the examination of this application in accordance with Title 37 of the Code of Federal Regulations §1.56. I hereby claim foreign priority benefits under Title 35, U.S. Code §119(a)-(d) or §365(b) of any foreign application(s) for patent or inventor's certificate, or §365(a) of any PCT international application which designated at least one country other than the United States, or §119(e) of any United States provisional application(s), listed below and have also identified below any foreign applications for patent or inventor's certificate having a filing date before that of the application on which priority is claimed:

Priority Claimed:
Yes [] No []

| (Application Number) | (Country) | (Day/Month/Year filed) | |
|----------------------|-----------|------------------------|----------------|
| | | | Yes [] No [] |
| | | | |
| | | | |

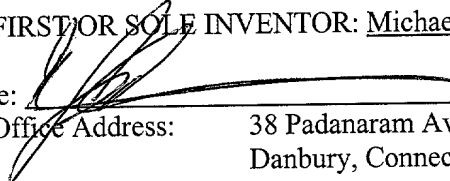
I hereby claim the benefit under Title 35, U.S. Code, §120, of any United States application(s), or §365(c) of any PCT International application designating the United States, listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States or PCT International application(s) in the manner provided by the first paragraph of Title 35, U.S. Code, §112, I acknowledge the duty to disclose information material to patentability as defined in Title 37, The Code of Federal Regulations, §1.56(a) which became available between the filing date of the prior application and the national or PCT international filing date of this application:

| (Application Serial Number) | (Filing Date) | (STATUS: patented, pending, abandoned) |
|-----------------------------|---------------|--|
| | | |
| | | |
| | | |

I hereby appoint the following attorneys: MANNY W. SCHECTER, Reg. No. 31,722; TERRY ILARDI, Reg. 29,936; CHRISTOPHER A. HUGHES, Reg. No. 26,914; EDWARD A. PENNINGTON, Reg. No. 32,588; JOHN E. HOEL, Reg. No. 26,279; JOSEPH C. REDMOND, Jr., Reg. No. 18,753; ROBERT P. TASSINARI, Jr., Reg. No. 36,030; DOUGLAS W. CAMERON, Reg. No. 31,596; KEVIN M. JORDAN, Reg. No. 40,277; STEPHEN C. KAUFMAN, Reg. No. 29,551; DANIEL P. MORRIS, Reg. No. 32,053; LOUIS J. PERCELLO, Reg. No. 33,206; JAY SBROLLINI, Reg. No. 36,266; ROBERT M. TREPP, Reg. No. 25,933; DAVID M. SHOFL, Reg. No. 39,835; LOUIS P. HERZBERG, Reg. No. 41,500; and PAUL J. OTTERSTEDT, Reg. No. 37,411, each of them of INTERNATIONAL BUSINESS MACHINES CORPORATION, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598; to prosecute this application and to transact all business in the U.S. Patent and Trademark Office connected therewith and with any divisional, continuation, continuation-in-part, reissue or re-examination application, with full power of appointment and with full power to substitute an associate attorney or agent, and to receive all patents which may issue thereon, and request that all correspondence be addressed to:

Frank Chau, Esq.
F. CHAU & ASSOCIATES, LLP
1900 Hempstead Turnpike, Suite 501
East Meadow, New York 11554
Tel.: 516-357-0091

I HEREBY DECLARE that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under §1001 of Title 18 U.S. Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

FULL NAME OF FIRST OR SOLE INVENTOR: Michael K. Gschwind Citizenship Austria
Inventor's signature:  Date: 1/26/2000
Residence & Post Office Address: 38 Padanaram Avenue #15A
Danbury, Connecticut 06811

FULL NAME OF SECOND JOINT INVENTOR: _____ Citizenship _____
Inventor's signature: _____ Date: _____
Residence & Post Office Address: _____

FULL NAME OF THIRD JOINT INVENTOR: _____ Citizenship _____
Inventor's signature: _____ Date: _____
Residence & Post Office Address: _____

FULL NAME OF FOURTH JOINT INVENTOR: _____ Citizenship _____
Inventor's signature: _____ Date: _____
Residence & Post Office Address: _____

FULL NAME OF FIFTH JOINT INVENTOR: _____ Citizenship _____
Inventor's signature: _____ Date: _____
Residence & Post Office Address: _____

FULL NAME OF SIXTH JOINT INVENTOR: _____ Citizenship _____
Inventor's signature: _____ Date: _____
Residence & Post Office Address: _____

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICANT(S): Michael K. Gschwind
SERIAL NO.: Unassigned
FILED: Concurrently herewith
FOR: **METHODS FOR RENAMING STACK REFERENCES IN
A COMPUTER PROCESSING SYSTEM**

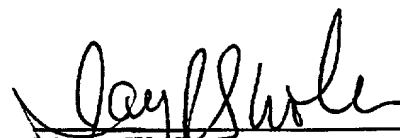
ASSOCIATE POWER OF ATTORNEY

Please recognize FRANK CHAU, Reg. No. 34,136; JAMES J. BITETTO, Reg. No. 40,513; FRANK V. DeROSA, Reg. No. 43,584; and GASPARE J. RANDAZZO, Reg. No. 41,528; each of them of F. CHAU & ASSOCIATES, LLP, 1900 Hempstead Turnpike, Suite 501, East Meadow, New York 11554 as associate attorneys in the above-mentioned application, with full power to prosecute said application, to make alterations and amendments therein, and to transact all business in the Patent and Trademark Office connected therewith.

Telephone calls should be made to Frank Chau by dialing (516) 357-0091.

All written communications are to be sent to Frank Chau, Esq., F. Chau & Associates, LLP, 1900 Hempstead Turnpike, Suite 501, East Meadow, New York 11554.

International Business Machines
Corporation
T.J. Watson Research Center
Route 134 and Kitchawan Road
Yorktown Heights, New York 10598


Manny W. Schecter
Registration No. 31,722
Jay P. Sbröllini
Registration No. 36,266
Attorney for Applicant(s)